STM32F4-Discovery
Tutoriel

Auteur du document
28 juin 2016

Table des matières

1 *Hello, world !* : faire clignoter une led

Utilisation du *cross-compiler* `arm-none-eabi-gcc`. La compilation utilise des *flags* spécifiques à l'architecture cible

```
ARMFLAGS = -mlittle-endian -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=
    fpv4-sp-d16
```

Le code source du premier programme fait clignorer une *led* sur la carte

```
#define GREEN_LED 12

int main(void) {

    system_init();

    /* Enable GPIOD periph clock */
    RCC->AHB1ENR  = RCC_AHB1ENR_GPIODEN;

    /* Set GPIOD pin to output mode */
    GPIOD->MODER |=1 << (GREEN_LED * 2);

    while (1){
        udelay (DELAY);
        GPIOD->ODR |= (1 << GREEN_LED);    /* led on */
        udelay (DELAY);
        GPIOD->ODR &= ~(1 << GREEN_LED);   /* led off */
    }
}
```

Le manuel utilisateur [**?**] de la carte indique que la led verte est connecté à la sortie GPIO *PD12*, ce qui correspond à la pin "12" du port *general purpose I/O* "D".

Lors de la compilation, le fichier est lié au fichier `startup_stm32f4xx.s` dont le rôle est d'initialiser les segments de données du programme (remplir le *bss* avec des valeurs nulles, déplacer les données en SRAM, etc.). La main est ensuite

donnée au programme en tant que tel. Celui-ci commence par appeler la fonction `system_init()` qui initialise la carte pour la rendre utilisable (cette fonction s'occupe principalement de configurer et de synchroniser entre elles les différentes horloges). Suite à cela, il faut explicitement *activer* l'horloge des périphériques que l'on souhaite utiliser, ici celle du port `GPIOD`.

```
RCC->AHB1ENR   = RCC_AHB1ENR_GPIODEN;      /* GPIOD Periph clock enable */
```

Ensuite, la pin doit être configurée en *output* en écrivant dans le registre `GPIOx_MODER`.

```
/* Set pin to output mode */
GPIOD->MODER |=1 << (LED_GREEN * 2);
```

Pour émettre un signal sur la pin du port GPIO, il faut écrire dans le registre `GPIOx_ODR`. Ceci permet par conséquent d'allumer ou d'éteindre la led

```
GPIOD->ODR |= (1 << led);    /* led on */
udelay (DELAY);
GPIOD->ODR &= ~(1 << led);   /* led off */
```

2 Déclencher une interruption avec un bouton poussoir

Ce programme va alterner le clignotement de la led verte (PD12) et de la led bleue (PD15) de la carte, en réponse à la pression du bouton bleu (PA0)

```
#define LED_GREEN       12
#define LED_BLUE        15
#define BLUE_BUTTON     0
```

L'initialisation des leds est similaire à la séquence vu précedemment

```
/* Enable GPIOD clock */
RCC->AHB1ENR   = RCC_AHB1ENR_GPIODEN;

/* Set pins to output mode */
GPIOD->MODER |=1 << (LED_GREEN * 2);
GPIOD->MODER |=1 << (LED_BLUE * 2);

/* Clear the leds (write low signal to the outputs) */
GPIOD->ODR &= ~(1 << LED_GREEN);
GPIOD->ODR &= ~(1 << LED_BLUE);
```

L'initialisation du bouton bleu est également très proche de ce qui a été vu jusque là

```
/* GPIOA Periph clock enable */
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

/* GPIOA0 pin set to input mode (00) */
GPIOA->MODER   &= ~(3 << (BLUE_BUTTON * 2));

/* Push-pull mode (0) */
GPIOA->OTYPER  &= ~(1 << BLUE_BUTTON);
```

```
/* Default (idle) state is at 0V. Set GPIO pin to pull-down (2) */
GPIOA->PUPDR   &= ~(3 << (BLUE_BUTTON * 2));   /* clear bits */
GPIOA->PUPDR   |=  (2 << (BLUE_BUTTON * 2));

/* Set GPIO Speed to high speed (2) */
GPIOA->OSPEEDR &= ~(3 << (BLUE_BUTTON * 2));   /* clear bits */
GPIOA->OSPEEDR |=  (2 << (BLUE_BUTTON * 2));
```

On observe toutefois certaines différences. Le registre `GPIOx_MODER` est configuré de manière à ce que la pin du bouton soit en *intput*. Le registre `GPIOx_PUPDR` permet de configurer la pin en *pull-up* ou en *pull-down*. Dans le cas présent, le circuit est ouvert par défaut, on configure donc la pin en *pull-down*. On note qu'il faut également configurer la vitesse du port. Plus le port va vite, plus il consomme de courant, ce qui peut être pénalisant pour un système alimenté de manière autonome. Dans notre cas, la carte est alimentée en USB, on peut donc utiliser la vitesse maximale.

Cette séquence d'initialisation indique que l'on utilise la pin associée au bouton en entrée, pour recevoir de l'information. Mais ceci ne suffit pas car il faut en plus déclencher une interruption quand le bouton est pressé. C'est ce que réalise le code ci-dessous

```
/* P<port>x interrupts are managed by the EXTIx interrupt line. Thus,
 * interrupts on BLUE_BUTTON are managed by EXTI0. EXTI0 can also manage
 * interrupts on ports PB0, PC0, PD0, etc. We must indicate in
    SYSCFG_EXTICR1
 * register that EXTI0 will only manage BLUE_BUTTON interrupts (p. 293) */
SYSCFG->EXTICR[0] &= 0xfffffff0;

/* Clear EXTI line configuration */
EXTI->IMR |=  EXTI_Line0;  /* Interrupt request from line x is not masked
   (1) */
EXTI->EMR &= ~EXTI_Line0;  /* Event Mask Register is masked (0) */

/* Trigger the selected external interrupt on rising edge */
EXTI->RTSR |= EXTI_Line0;  /* Rising */
EXTI->FTSR &= ~EXTI_Line0;  /* Clear falling */

/* Set the IRQ priority level (in the range 0-15). The lower the value, the
 * greater the priority is. The Reset, Hard fault, and NMI exceptions, with
 * fixed negative priority values, always have higher priority than any
    other
 * exception. When the processor is executing an exception handler, the
 * exception handler is preempted if a higher priority exception occurs.
 * Note: 'EXTI0_IRQn' stands for EXTI Line0 Interrupt */
NVIC->IP[EXTI0_IRQn] = 0;

/* Enable the Selected IRQ Channels */
NVIC->ISER[0] = (uint32_t) 0x01 << EXTI0_IRQn;
```

Les interruptions externes sont multiplexées, mais non pas en fonction du port GPIO mais en fonction du numéro de pin. Par exemple, les interruptions en provenance des pins PA3, PB3, PC3, PD3, etc. transitent toutes par la ligne d'interruption `EXTI3`. Dans notre cas, le bouton bleu est sur la pin 0 du port GPIOA, l'interruption sera donc associée à la ligne `EXTI0`. L'instruction suivante, à la fin du fichier, met en oeuvre les interruptions pour cette ligne

```
NVIC->ISER[0] = (uint32_t) 0x01 << EXTI0_IRQn;
```

Il faut configurer le registre `SYSCFG_EXTICR` pour indiquer que les interruptions arrivant sur la ligne `EXTI0` proviennent de la pin `PA0`. Le démultiplexage est réalisé de manière statique

```
SYSCFG->EXTICR[0] &= 0xfffffff0;
```

Le processeur supporte deux types d'interruptions : les interruptions à proprement parler et les évènements (*event*). Les évènements sont une signalisation envoyée au processeur dont le but semble essentiellement être de sortir le processeur d'un mode de veille, nous ne les utiliserons donc pas ici. On autorise (démasque) les interruptions sur la ligne `EXTI0`

```
EXTI->IMR |=  EXTI_Line0;  /* Interrupt request from line x is not masked
    (1) */
EXTI->EMR &= ~EXTI_Line0;  /* Event Mask Register is masked (0) */
```

L'interruption est déclenchée lors d'un front montant (le signal passe de "bas" à "haut"), c'est-à-dire lorsque l'on ferme le circuit en pressant le bouton

```
EXTI->RTSR |= EXTI_Line0;  /* Rising */
EXTI->FTSR &= ~EXTI_Line0;  /* Clear falling */
```

Enfin, comme vu précedemment, on active les interruptions pour la ligne `EXTI0`. Mais avant cela, on peut attribuer une priorité à ce type d'interruptions. La priorité à une valeur sur 4 bits allant de 0 à 15. La valeur 0 correspond à la plus haute priorité. Si deux interruptions avec des priorités différentes sont déclenchées au même moment, celle avec la plus haute priorité sera exécutée. A noter que si le processeur exécute un *handler* d'interruption et qu'une interruption plus prioritaire survient, cette dernière préempte le processeur

```
NVIC->IP[EXTI0_IRQn] = 0;
NVIC->ISER[0] = (uint32_t) 0x01 << EXTI0_IRQn;
```

Le rôle du *handler* d'interruption est ici de changer de led. Il doit absolument acquitter l'interruption sans quoi elle se répète indéfiniment

```
void EXTI0_IRQHandler(void)
{
/* Clear the led */
GPIOD->ODR &= ~(1 << currentLED);

/* Change current led */
currentLED = (currentLED == LED_GREEN) ? LED_BLUE : LED_GREEN;

/* Clear Pending Request bit to acknowledge the interrupt (this bit is
    cleared by programming it to 1 ! */
EXTI->PR = EXTI_Line0;
}
```

Le nom de ce *handler* n'est pas choisi au hasard. Il est lié au fichier `startup_stm32f4xx.s` qui contient la table des vecteurs d'interruptions du processeur. Chaque entrée de la table contient une valeur par défaut qui peut être surchargée lors de l'édition de lien grâce à une directive spéciale de `gcc`.

Références

[1] UM1472 User manual. *Discovery kit with STM32F407VG MCU*. STMicroelectronics, February 2016.